ADVANCED END-TO-END SIMULATION FOR ON-BOARD PROCESSING (AESOP)

Alan S. Mazer

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, California 91109

1. Introduction

Developers of data compression algorithms typically use their own software together with commercial packages to implement, evaluate and demonstrate their work. While convenient for an individual developer, this approach makes it difficult to build on or use another's work without intimate knowledge of each component. When several people or groups work on different parts of the same problem, the larger view can be lost. What's needed is a simple piece of software to stand in the gap and link together the efforts of different people, enabling them to build on each other's work, and providing a base for engineers and scientists to evaluate the parts as a cohesive whole and make design decisions.

AESOP (Advanced End-to-end Simulation for On-board Processing) attempts to meet this need by providing a graphical interface to a developer-selected set of algorithms, interfacing with compiled code and standalone programs, as well as procedures written in the IDL and PV-Wave command languages. As a proof of concept, AESOP is outfitted with several data compression algorithms integrating previous work on different processors (AT&T DSP32C, TI TMS320C30, SPARC). The user can specify at run-time the processor on which individual parts of the compression should run. Compressed data is then fed through simulated transmission and uncompression to evaluate the effects of compression parameters, noise and error correction algorithms.

The following sections describe AESOP in detail. Section 2 describes fundamental goals for usability. Section 3 describes the implementation. Sections 4 through 5 describe how to add new functionality to the system and present the existing data compression algorithms. Sections 6 and 7 discuss portability and future work.

2. Design Goals

A few goals are central to the design of AESOP. AESOP must:

1. Be usable enough that scientists and system designers can experiment with their data with little instruction. There must be clear visual feedback as applications execute. The user must be able to easily display algorithm data using a variety of display types.

2. Be easy to augment. It should be easy to integrate executables for which source is unavailable, as well as code written in compiled languages such as C and FORTRAN. Non-programmers should be able to use a high-level interpreted language to add capabilities.

3. Rely on outside development when such is commonly and cheaply available. It should provide for the integration of commercial packages as much as possible.

4. Isolate itself from applications; changes to AESOP must not require that applications be rebuilt or otherwise modified.

5. Provide complete error handling. AESOP must be prepared to handle internal errors, user errors and errors in applications, in a useful way, preserving the current state and providing the user options as much as possible.

6. Coexist well with other executing software. It should be efficient and flexible in use of screen space and other system resources.

7. Be user-customizable in look. The user should be able to choose cosmetic features such as user interface colors, as well as operational defaults, such as which types of displays are automatically enabled.

3. Implementation

The AESOP implementation assumes two simple concepts: **modules**, compiled or interpretable code which performs specific computations, and **algorithms**, module sequences used to implement complete applications. The following sections describe these two concepts in more detail, and then show how they provide a basis for the complete system.

3.1. Modules and Algorithms

Each AESOP module, compiled or interpreted, has a usage type and some number of input and output arguments. Input modules are used to read in files from disk or bring other data into the system which the user can't practically enter from the keyboard. Compute modules perform computational tasks. Output modules are selected at run-time by the user and perform data display. Arguments also have usage types. An input argument is one read by the module; an output argument is a value or data item that the module generates. Update arguments are both read and modified by the module. Each argument also has a data type, as summarized in Table 1.

Table 1 – AESOP data types

char	char_1d	char_2d
short	short_1d	short_2d
int	int_1d	int_2d
float	float_1d	float_2d
double	double_1d	double_2d
string	string_1d	string_2d
kwd	kwd_1d	kwd_2d

An AESOP algorithm is a sequence of compute modules where the inputs for each module are taken either from the user or from the output of a previous compute module. Algorithms are typically a mixture of compiled and interpreted modules.

3.2, The Dictionary Interface

Figure 1 shows an overview of AESOP implementation. Sections 3.2 through through 3.4 will discuss the major components, beginning with the dictionary interface and continuing with code execution and the GUI.

Dictionaries are ASCII files listing available **modules** (compiled routines, binary executables, interpretable procedures) and **algorithms** (module sequences designed to perform common tasks), AESOP looks for one standard dictionary, "stdlib.dict", to contain generally useful routines for output display, local file formats, etc. Users may define any number of other dictionaries to describe modules and algorithms in specific application areas. AESOP looks for dictionaries in the local directory, with the AESOP executable, and in other directories specified by the user using the AESOP_APPLDIRS environment variable. Dictionaries can be re-read without leaving AESOP to gain across to newly-defined or modified algorithms and modules. Dictionaries can also contain graphics directives specifying how an algorithm is displayed on the screen, including labels and boxes. Dictionary entries have several formats depending on whether they are defining a compiled module, an interpreted PV-Wave module or an algorithm.

Entries for compiled modules have the form:

module type name:label: pathname

PV-Wave module.s are defined similarly, but with the. module inputs and default values following the pathname. Entries for interpreted PV-Wave modules have the form:

module_type name:label :pathname:
arg_use_type$_1$ arg_data_type$_1$ arg_label$_1$ [= default],
arg_use_type$_2$ arg_data_type$_2$ arg_label$_2$ [= default], . . .
arg_use_type$_n$ arg_data_type$_n$ arg_label$_n$ [= default]

The first line of the entry is similar to the entry for the compiled-module. Subsequent lines list parameters, separated by commas, where each parameter has a use type, data type and prompt. Initial values may be specified by following the prompt with an equal sign (=) and the value. Scalars are considered user options automatically; higher-dimensioned parameters are retrieved from previously-executing modules. Type conversions are implicit.

Dictionary entries for algorithms have the basic form:

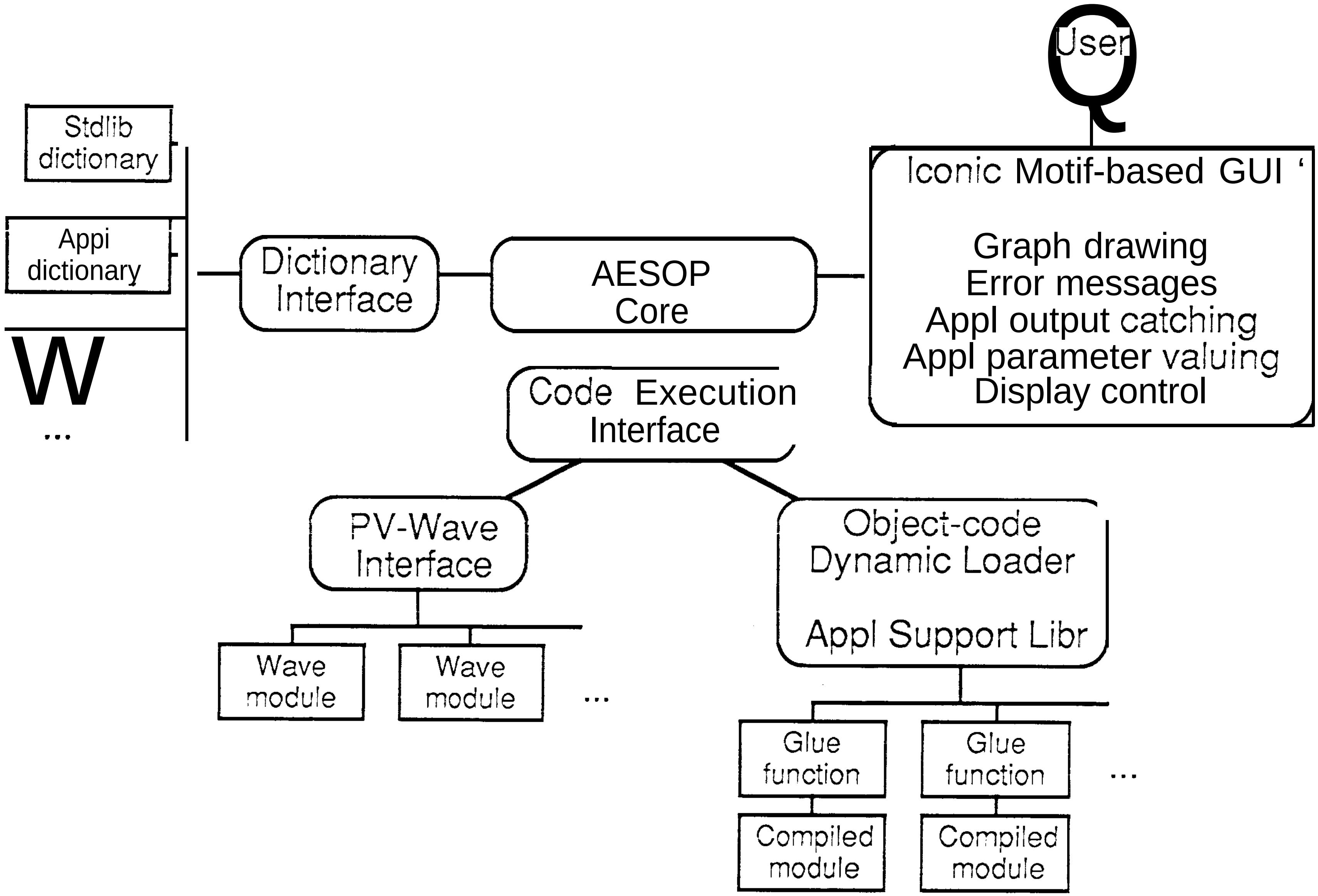
algorithm name:label:module$_1$ module$_2$ . . . module$_n$

Extensions to this basic syntax allow the user to group modules in labeled boxes and to lay these boxes in any direction.

3.3. The Code Execution Interface

AESOP provides access to two different types of modules: interpreted modules written in the PV-Wave command language and compiled modules written in C or another high-level language. Both types of module.s have "glue functions" which are called by AESOP and call the module code in turn. This approach isolates the details of executing application code from AESOP internals.

Implementation Overview



in the case of compiled code, glue. functions are programmer-written and allow AESOP to call executables for which source code is unavailable, as well as routines written in languages other than C. The glue function, written in C, creates local storage for use by the function and defines parameters in a manner AESOP can understand. AESOP calls these glue functions using dynamic loading, further isolating application routines from AESOP itself. The parameter definition interface is simple, using keywords and program-callable functions for optimal capabilities, allowing the interface to be extended in the future without requiring modification of currently-integrated code. Glue functions for compiled modules take a single argument, an initialization flag. When an algorithm is selected, AESOP calls the glue function for each compiled module in the algorithm with the initialization flag set to 1. At this time each module uses the AESOP def() function to describe its parameters where def() is defined:

```
def(char *prompt, enum use_type use, enum data_type type,
   void *local_addr, char *kwds[], int num_kwds, int option1,
   int option2, ..., 0)
```

The glue function will be called a second time, with the initialization flag 0, when the module is actually executed. The kwd data types provide a simple way to restrict the user's choice of values. Glue functions can indicate an error in either their initialization or execution parts by returning -1, causing AESOP to stop algorithm execution with that module.

For PV-Wave modules, a generic glue function is supplied by AESOP. Since PV-Wave modules have their parameters defined in the dictionary, their glue function need only be called at execution time, where it creates temporary files needed to communicate with PV-Wave, instructs PV-Wave to read necessary data, and invokes the PV-Wave procedure. Module parameters listed in the dictionary and valued by the user before the run are passed in as arguments to the procedure. The AESOP-Wave interface uses temporary files and PV-Wave's cwavec() facility. The AESOP-Wave interaction is transparent to the developer and user.

When an algorithm is loaded, AESOP automatically matches up non-user-specifiable parameters. It does this by comparing the names of module outputs with the names of inputs from subsequent modules and assigning to each possible match-up a score. This scheme will probably need to be refined in the future. At the moment, close attention must be given to an algorithm in development to make sure AESOP is attaching inputs to outputs as expected. AESOP uses dimensionality and data type to reduce the potential for error. Nevertheless, simple generic names are best, for example, "output image." rather than "decompression output". In the latter case, a subsequent module, expecting "input image" might get connected up with some other "image" in the system, rather than the more ambiguous "decompression output". Once all the connections have been made, AESOP uses the PV-Wave or dynamic loading interface as necessary to execute each module in turn. AESOP ensures before each module is executed that the inputs to the module are available, either because the user explicitly specified them or because they were generated by a previous module in the algorithm. Signal handlers are installed to catch memory usage errors in applications. If AESOP detects such an error it stops execution of the module, restoring itself to its state before execution started.

3.4. The GUI

The usability goals described in Section 2 are met in part by a graphical interface. Most user interactions can be done with the mouse. The current status of the system is graphically displayed. Options prohibited in a specific context are hidden until needed to avoid confusion. The implementation is divided into 5 general parts: graph drawing, error messages, application output catching, application parameter valuing and display control.

The graph drawing section presents algorithm selected as dataflow diagrams, Graph drawing is done using X11/Motif, with application modules represented by boxes and connected with arrows in a single-stream pipeline. Modules may be grouped and groups labeled. Groups may be oriented in any direction, clearly distinguishing different parts of an algorithm. Grouping, labeling and orientation are optional and taken from the algorithm specification in the dictionary. When algorithm execute, module boxes are highlighted to show progress. Since for large algorithms the graph area may not be large enough to show all the modules, the graph area scrolls itself to keep the currently executing module visible.

The error messages section alerts the user to AESOP-discovered error conditions using popup windows. AESOP detects 39 different error conditions, including fatal memory usage errors in application modules. AESOP shows a popup window describing the condition and then waits for user acknowledgement before continuing. Error messages printed by an application module are also displayed in popup windows.

Non-error output from an application module is caught and optionally displayed in its own window, When a module tries to send informational messages to the user, AESOP grabs that output and, if the user has requested diagnostic output, displays it in a window created for that purpose. Otherwise the output is discarded. AESOP can maintain a separate. window for each module, and switch between them as the different modules execute. This capability allows the user to choose which parts and how much of the execution details to view, and simplifies debugging during module development.

The application parameter valuing section allows the user to give values to optional and required module parameters using popup windows. Both interpreted and compiled modules may take parameters. The user specifies a value for a module parameter using the, pulldown menu attached to the module in the graph. AESOP lets the user enter scalar numerical quantities or choose items from lists using the keyboard. For larger parameters like input images the user selects a module, to use to read in the required data. Such modules are typically defined in the standard library but are otherwise similar to application modules.

Finally, AESOP allows the user to monitor module inputs and outputs using a variety of display types. When AESOP starts it builds a list of all output modules listed in the dictionaries. It then sorts the modules based on data type and the dimension of the primary input(s), where a primary input is defined as an input such that no other input has a larger number of dimensions, When the user requests display of a module input or output using a module's menu, AESOP allows the user to select a parameter to display and then presents a list of output modules suitable for displaying that particular type of value. Alternatively the user can add a display using the **Displays** menu. AESOP allows the user to specify the dimensionality of the. data and the type of display to create using the menu, and then presents a list of module parameters displayable with that type of output module.. Since some display modules will take inputs other than the data to display, AESOP prompts the user for needed information; in the case of non-scalar inputs, it offers choices from among the data items currently available in the system. These capabilities are provided automatically by AESOP and do not depend on the algorithm writer. The **Displays** menu also allows users to change or remove displays. PV-Wave has been used to implement most of the current output modules.

Figures 2 and 3 show AESOP adding noise to a JPEG-compressed image and the resulting output with no error correction.

4. Programming Environment

Adding functions or subroutines written in C, FORTRAN and other compiled languages requires only writing the glue function and adding the name and object file pathname to a dictionary. Glue functions for compiled modules have two pare: the initialization part which defines parameters using AESOP's def() function, and an execution part to call the compiled function. Glue functions should return -1 on discovering a fatal error anti 0 otherwise. Error messages should be written to stderr and informational messages to stdout. The dictionary entry for the DCT compute module declares the type. of the module, its name, the label to use on the graph, and the pathname of the glue-functicm object:

```
compute_module jpeg_dct:DCT:lib/rpc.so
```

The glue function must be compiled and linked with the functions it calls into an executable with a ".so" extension. For SunOS one would use:

```
acc -c -pie glue funcs.c
ld -0 library.so glue funcs.o funcs to add.o
```

Generally useful functions should go into the standard library ("stdlib.diet"). Other functions can be listed in application dictionaries. Once the module, has been specified in one or the other type of dictionary it's available for use.

Adding code from l'V-Wave and other colllllal~d-lillc-based packages is similar to adding compiled code., except that parameters arc declared in the dictionary rather than using a glue, function:

```
output_module flick2:Alternate Two Images:flick2.pro:
     input u_char_2d First Image, input u_char_2d Second Image,
     input int Iterations=20, input float Wait=0.3
```

Algorithms arc added by simply defining them in the (dictionary as an ordered list of module names:

```
algorithm jpeg:JPEG:"jpeg_dct jpeg_quant jpeg_huff jpeg_decomp
```

The dictionary syntax allows the user to group modules in labeled boxes and to lay these boxes in any direction. A group is introduced using a vertical bar (1) followed immediately by tile label for the group, a direction indicator (>, <, ^ or !), a list of space-separated modules forming the group, and the direction indicator again. The algorithm shown in Figure 2. was defined using:

```
algorithm jpegendtoend:JPEG End-to-end:
     |Compress>jpeg_dct jpeg_quant jpeg_huff>
     |Xmit!packet segment addnoise unsegment unpacket!
     |Decompress<jpeg_decomp<
```

S. Data Compression Applications

Application development for AESOP so far has centered on data compression, but includes simulation of flight-to-ground downlinks. Thus there arc application modules not only for various types of compression (JPEG, Rice, one- anti two-dimensional wavelet compression) but also for packetization, segmentation, channel coding and noise simulation, providing a true end-to-end view from in-flight data acquisition to the reception of transmitted data on the ground. Supporting the end-to-end simulation of compressed data transmission are a number of computational capabilities (packetization, segmentation and channel coding, and noise simulation) as well as output types.

The packetization routine takes compression output and a set of packet lengths in bits, and breaks the output into packets at the specified bit boundaries. Currently, variable length packets are formed such that each packet holds 8 lines of compressed image data. This approach simplifies recovery should an entire packet be lost since the location of a packet in an output image can be coded in the header, and the break is guaranteed not to occur in the middle of a pixel. An inverse procedure takes incoming packets and recombines them into a single bit stream for decompression.

Because channel coding requires fixed-length chunks of input data, packets are themselves grouped into interleaved segments of uniform length; segments are packed into frames. The interleave factor is an option with a default value of 8. Segmentation currently uses Reed-Solomon coding for optional error correction. The inverse procedure unencodes the data and restores the original input packets. Some diagnostic information (error counts, frame statistics) is available using **Show diagnostics** on the module's menu.

A noise simulation module takes compressed, packetized, segmented data and flips bits on a random interval. The user can specify the mean number of bits between errors, or turn off noise simulation altogether. Better noise models are being developed.

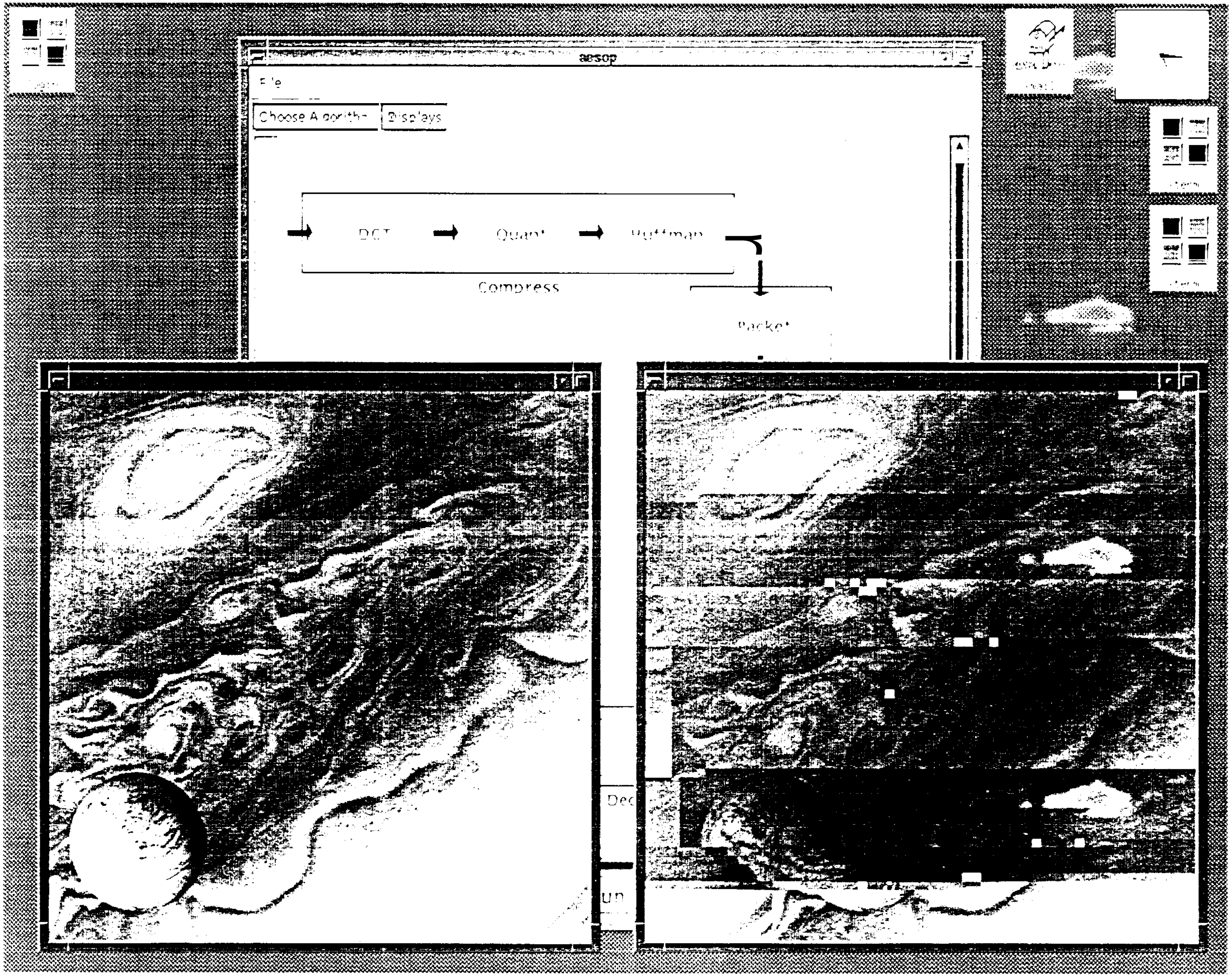
In addition to many output modules in the standard library for reading, writing and displaying various data types, of special interest for data compression algorithm are "Showboth", which allows a user to see two different images side by side, "Flick2", which alternates two images rapidly in the same window using a user-chosen interval and number of iterations, and "Imagediff", which displays the difference of two images using a user-chosen multiplication factor. These are currently restricted to byte input images. Other modules compute signal-to-noise ratios for most vector and image data types.

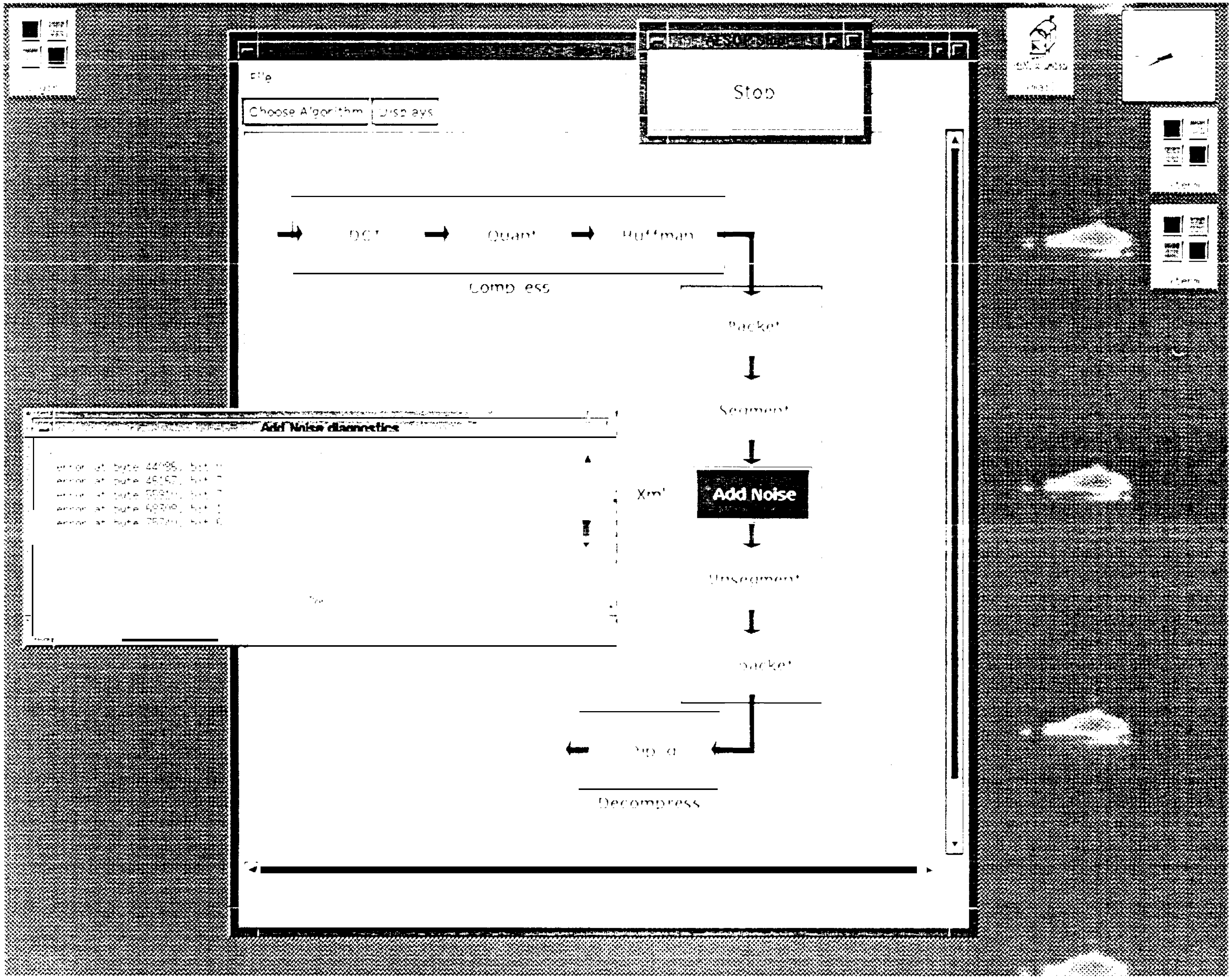
6. Portability

AESOP currently runs on Sun SPARCstations using SunOS 4.1.3 and Motif. While PV-Wave is not required, support for it is built in and the current dictionaries use it for image display. Operating system dependencies are minimal. AESOP is written in ANSI C. AESOP uses dynamic loading to execute compiled modules, which is available on AIX 3.2, HPUX 8.0 and VMS 5.0 in addition to SunOS.

7. Future Work

The foundation is in place, but work remains to be done. AESOP currently relies heavily on PV-Wave for output display; other packages need to be integrated for portability. More output types, particularly for one-dimensional data, need to be implemented. Support for application-defined data structures would be useful. Some applications may have trouble with AESOP's redefinition of the C `write()` routine. Determination of graph connectivity will eventually need enhancement. More control over output displays needs to be added.


aesop
File
Choose Algorithm
Displays
DCT
Quant
Huffman
Compress
Packet
Dec
Un


File
Choose Algorithm
Stop
DCT
Quant
Huffman
Compress
Packet
Segment
Add Noise
Xmt
Unsegment
Decompress
Add Noise diagnostics